フラクタル符号化における並列計算の検討
－GPU,MPIによる実装と評価－

（平成25年11月28日改訂）

鶴見　智*

（2013年11月28日受理）

１．はじめに

近年，非構造化データ，いわゆるビッグデータの処理技術に注目が集まっている．その中で，画像ビッグデータからの類似画像検索，ビデオシーン検索等の効率的な手法が求められている．その際，対象となる画像や動画のデータ量は膨大なものになるため，高圧縮符号化されていることが必要であり，可能ならば復号せずに符号データを直接使用できることがのぞましい．以上の，観点で見た場合，フラクタル符号化[1][2][3]は最適な符号化法の一つと言える．

フラクタル符号化は，高圧縮率，高速復号等の特長を持つ一方，符号化時間が膨大になるという欠点もある．符号化アルゴリズムの主要部分は，画像の中からもっとも自己相似性の近い画像ブロック・ペアを見つけることにあるが，その計算量が膨大になることが，符号化時間がかかる主な理由である．

一方，アルゴリズムの単純さと同種処理の繰り返しの多さは並列計算向きのアルゴリズムといえる[4]．最近のマルチコア・プロセッサの普及や高性能のGPU(Graphics Processing Unit)の低価格化は，フラクタル符号化の並列アルゴリズムの標準化を検討する時期に来ていること，フラクタル符号化による静止画・動画のリアルタイム符号化の実現が可能になりつつあることを意味している．

本研究の目的は，GPU によるフラクタル符号化の実装を通し，並列化フラクタル符号化アルゴリズムの標準化の検討をすることである．また，マルチコア・プロセッサでの並列化を行う MPI(Message Passing Interface)による実装との比較検討も行う．

２．関連研究

GPUを用いたフラクタル符号化の関連研究としては 先駆的なU.Erraの研究[5]，医用画像へ適用した研究[6]，フラクタル動画符号化を検討した研究[7]がある．これらの多くは実験的実装であり，実用的な適応的なブロック分割を考慮しておらず，並列化にあたってのブロック分割に伴うメモリ処理等の検討もされていない．本研究は，実用的なアルゴリズムとして知られる４分木分割アルゴリズムによるフラクタル符号化をGPUとMPIで実装し，比較検討を行い，標準化の問題点を考察する．

３．フラクタル符号化

フラクタル符号化は，画像に存在する自己相似性を利用して，自身の近似画像を生成するブロック符号化の１種である．これは縮小アフィン変換の集合である反復関数系(Iterated Function System, IFS)によって実現される．すなわち画像に対して IFS を決定することがフラクタル符号化である．フラクタル符号化アルゴリズムの概念を図１に示す．

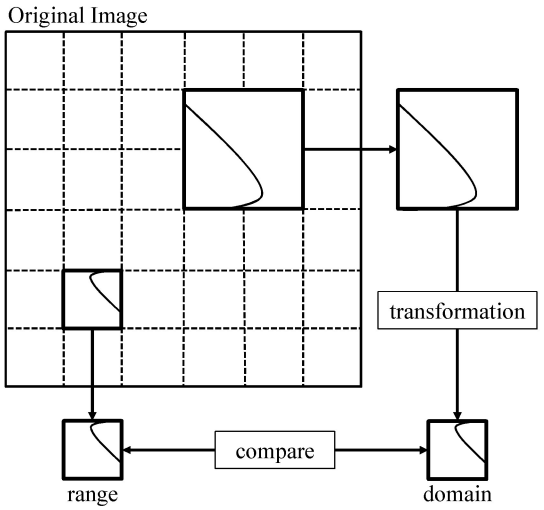

図１　フラクタル符号化

*電子情報工学科

基本的な符号化手順は以下のとおりである.

1) 原画像を互いに重複しない $N \times N$ 画素のレンジブロック（range）に分割する．また，同様に原画像から $2N \times 2N$ 画素のドメインブロック（domain）を取り出す.
2) それぞれのドメインブロックに対し，対称変換を行う．対称変換は，ブロックを 0 度，90 度，180 度，270 度回転させた後，対角線に対する反転を行ったもので，各回転に対し 8 種類存在する．次に対象のレンジブロックと同じブロックサイズになるよう縮小変換を施す．2×2 画素ごとに平均化することによってサイズを縮小する．（ドメインプールを作る）
3) 2)で作られた全ドメインブロックと対象のレンジブロックを比較し，以下の 2 乗和誤差が最小になるように輝度スケーリング s と輝度シフト o の変換係数を決定する.

$$\sum_i \sum_j (s \bullet d_{ij} + o - r_{ij})^2 \qquad (1)$$

ここで，d_{ij} は位置 (i, j) でのドメインブロックの画素値，r_{ij} は位置 (i, j) でのレンジブロックの画素値である.

4) 全レンジブロックに対し同様の処理を行う.
5) 各レンジに対応するドメインの位置，対称変換およびを輝度スケーリング，輝度シフトを符号とし保存する.

実用的なフラクタル符号化として，Fisherは４分木分割フラクタル符号化アルゴリズムを提案した[1]．処理手順は1)～5)と同様だが，レンジブロックとドメインブロックの2乗和誤差が最小となる組み合わせを探索する過程で閾値を設ける．閾値は任意に設定可能であり，これを満たさない場合は当該レンジブロックのサイズを4分割（再分割と呼ぶ）し，その全てに対して再び探索を行う．再分割が行われるごとにレンジブロック，ドメインブロックのサイズは半分になる．以上の手続きを誤差が閾値を満たすか，あるいは指定したブロックサイズになるまで反復する．このアルゴリズムの流れ図を図２に示す．閾値を小さくすることあるいは再分割の深さを深くすることで画質が向上するが，計算量が増大し符号化時間が増える.

一方,復号化は任意の画像に対し,変換係数を用いて全レンジブロックを生成し,こうして作られた画像に対して同様の操作を繰り返すことで行う.この繰り返しは10回程度で収束するため,復号は極めて高速である.

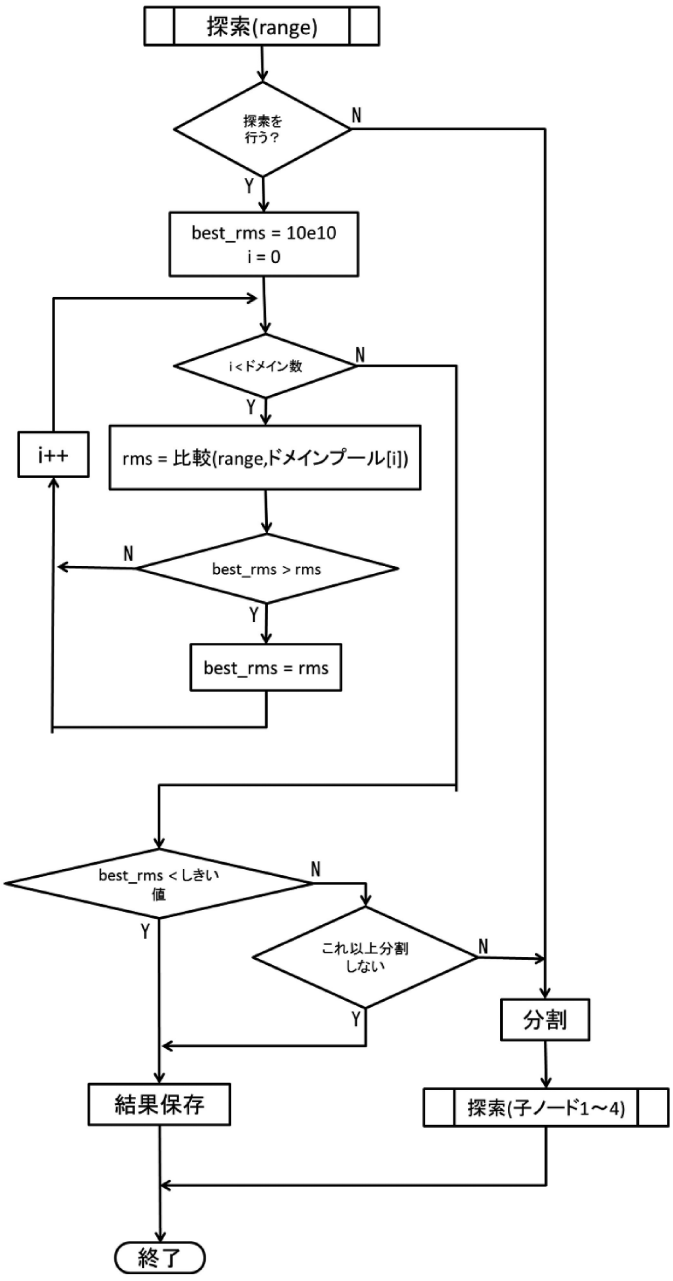

図２　４分木分割フラクタル符号化の流れ図

４．GPU

コンピュータで演算機能を担うのは CPU(Central Processing Unit)であるが,近年のゲームや映像の高画質化とともに,グラフィックス処理が膨大になってきた.これに対処するため通常グラフィックス処理は CPU と独立したビデオカードで行なっている.このビデオカード上に搭載されているグラフィックス処理専用に開発されたユニットが GPU(Graphics Processing Unit)である.この GPU を汎用数値計算に用いる技術を GPGPU(General Purpose Computing on GPU)と呼び, NVIDIA 社が開発環境 CUDA を提供した 2000 年以降急速に進展した.

GPU は以下の特徴を持つ[8].

(1) 各画素においての処理はそれぞれ独立に行える.

(2) 出力された画像データは保存しておく必要がない.

これによって大量のデータを複数のプロセッサで同時かつ並列に処理することができる. 今回は NVIDIA 社製 GPU を実装に使用する. 図３に NVIDIA 社 GPU と CPU の演算性能の推移を示す. CPU の伸びが鈍いのに比べ, GPU は今後も大幅な性能向上が見込まれる.

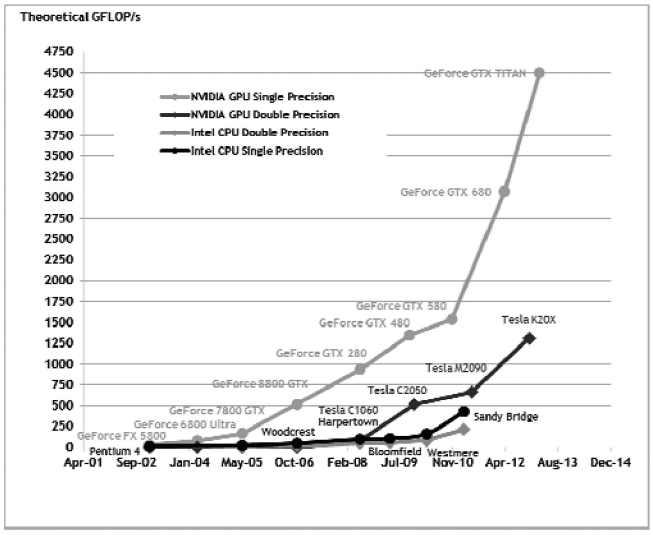

図３　GPU と CPU の高速化の推移 (NVIDIA CUDA Programming Guide 5.5[9]より引用)

図４にビデオカードの内部の簡略図を示す. GPU の内部には CUDA コアと呼ばれる演算装置があるが, それらは１個ずつ独立しているのではなく, 一定数のストリーミングマルチプロセッサ(SM)で統合されている. Kepler アーキテクチャである Tesla K20 は SM は 13 個, CUDA コアは 2,496 個からなる.

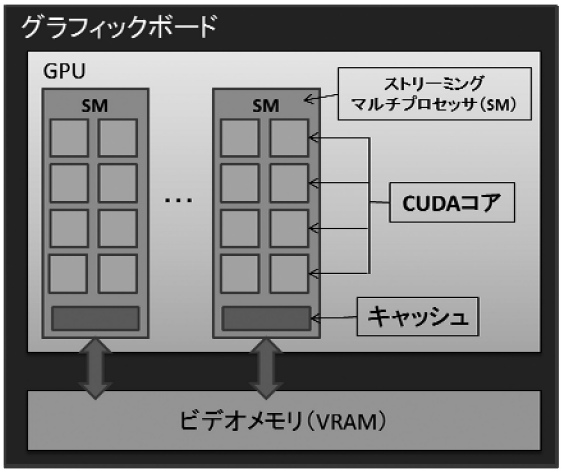

図４　ビデオカードの内部構造

メモリ GPU とビデオメモリは「メモリインターフェース」で接続されており, CPU～メインメモリ間より数倍大きなメモリバンド幅で接続されているので高速に伝送できる. しかし, グラフィクボードにはグローバルメモリ, ローカルメモリ, シェアードメモリ, コンスタントメモリ等, 何種類かのメモリがあり, 容量の大小, アクセス速度に違いがあるので, 効率よく計算させるにはそれらを考慮したプログラミングが必要となる.

CUDA プログラミングの流れを図５に示す[8]. ここでは, CPU やメインメモリ側をホスト, ビデオメモリ側をデバイスと呼び区別している.

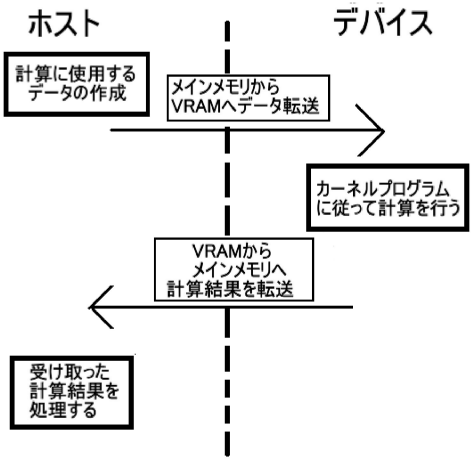

図５　CUDA プログラミングの流れ

５. MPI

MPI(Message Passing Interface)は並列プログラミングのための規格であり, C や Fortran のライブラリとして提供される. MPI による並列プログラミングでは複数のプロセッサ（コア）が同時に同じプログラムを複数個実行する. プログラムの実行単位をプロセスと呼び, ひとつのコアはひとつのプロセスを実行する. 通常, メインとなるプロセスがデータを各プロセスに配分し, 各プロセスは独立して処理を実行し, 最後にメインプロセスがデータを集約する. すなわち SPMD(Single Program Multi Data)が MPI の基本である. 通信関係のサブルーチンとして重要なものは, １対１通信, 集団通信である. 図６に基本的な通信である１対１通信を示す. MPI はネットワークを介したマルチコアにも対応しているが, 実装にあたっては通信コストを考慮に入れておく必要がある.

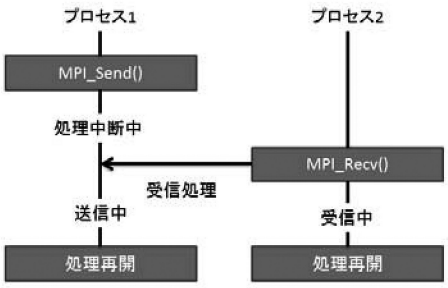

図６　１対１通信

６．GPUによるフラクタル符号化の実装

GPU によるフラクタル符号化の実装には以下の２つの方法を検討した.

方法 A：

フラクタル符号化を GPU で実装する最も単純な方法は，各レンジとドメインの２乗和誤差の計算をカーネル関数として各コアに振り分けるやり方である（図７参照）.

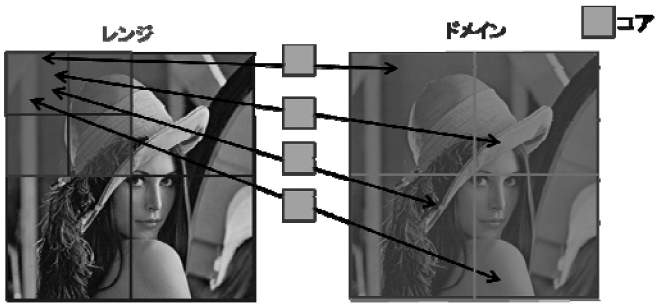

図７　GPU での実装

この場合，レンジごとに転送を行うとデータの転送回数が増え，高速化は期待できない．そこで，各レンジの結果をその都度転送せずにいったんシェアードメモリに保存し，全スレッドの比較が終わったあとで最適な結果を転送することとする．このとき最適な結果すなわち２乗和誤差の最小値を求めるために並列リダクションアルゴリズムを用いる.

方法 B：

あらかじめレンジ１つ分のピクセル値を行列の１行に並べていき，全レンジからなる行列（レンジ行列と呼ぶ）を作る．同様に全ドメインからなるドメイン行列を作る．このとき２乗和誤差計算はレンジ行列とドメイン行列の行列積計算に帰着する．行列積の計算は CUDA で高速に実行できるため，符号化時間の大幅な短縮が可能となる.

どちらの方法も，画像のアクセスはテクスチャメモリを使用せず，コンスタントメモリから行うことで高速に処理するようにする.

７．MPIによるフラクタル符号化の実装

プロセスは４の累乗とする．４プロセスでの実装は以下のように行う．まずプロセス１で画像を読み込み，続いて画像を４分割し，４つのプロセスに配分し，それぞれのプロセスで４分木分割フラクタル符号化を行う．すべてのプロセスが終了したら，プロセス１に結果を伝送し，符号を保存して終了する．このとき，ドメインプールは元画像で分割したものを使う.

８．実験結果

実験は，CPU：Intel Core i7 2600(3.40GHz, 4 コア)，メモリ 8GB，GPU：NVIDIA GeForce GT545(1.74GHz, 144 コア,メモリ 1GB)，CUDA バージョン 4.2 で実装し，行った．また，MPI は Mpich2-1.4.1pl を用いた．符号化実験に使用した画像は，Lena，Mandrill，Pepper（すべて 512×512pixcel，8bpp グレースケール）である．レンジの再分割は 64×64～4×4 までとした.

符号化時間を測定した結果を表１に示す．ただし，実行時間は GPU，MPI ともに 10 回の平均をとっている.

方法	Lenna	Mandrill	Pepper
CPU	5.041	5.675	5.466
GPU(方法A)	3.663	4.049	3.963
GPU(方法B)	2.353	2.601	2.527
MPI(4コア)	1.447	1.550	1.503

表１　符号化時間の比較[秒]

GPU（方法 A）では約 72%に，GPU（方法 B）では約 46%に高速化が達成できている．Lenna や Pepper に比べ Mandrill の方が高速化の割合が大きいのは,エッジ情報の多い Mandrill では再分割が多くなり,その分並列化の効果がでていると考えれる．MPI は約 28%に高速化されているが，測定値は画像分配後から全プロセスが終了するまでの実行時間であり，通信の時間は含まれていないため GPU と単純に比較できない．特に，複数台のマルチコア・マシンをイーサネットで接続して実行する場合，通信時間が大幅に増え，並列化の効果が失われる可能性がある.

比較のために表２に Lena に対する GPU でのカーネル実行時間とメモリ転送時間を示す．ただし，計測は１回の実行時の値である.

方法	実行時間	カーネル	メモリ転送
GPU(方法A)	3.734	3.64	0.015
GPU(方法B)	2.377	0.789	0.033

表２　GPU でのカーネル実行とメモリ転送の時間[秒]

方法 B は方法 A に比べ，メモリ転送時間はかかっているが，カーネルでの実行時間は大幅に短縮している．これは，GPU による行列計算の高速化の効果とみられる．

以上より，並列リダクションアルゴリズムと GPU に最適化された行列計算アルゴリズムを用いることで，GPU でリアルタイム符号化を実現できることがわかった．このとき,テクスチャメモリでなくコンスタントメモリを使用することで高速化が保証される.また,メモリ転送速度の測定から,メモリ転送速度の改良ができれば，さらに高速化も可能であると考えられる．

９．まとめ

フラクタル符号化をGPUとMPIで実装し，比較評価をした．GPUでは，グラフィックボードの高速なメモリを使用することに注意し，GPUに適したアルゴリズムに改良することで，静止画については十分リアルタイム符号化が可能であることを示した．また，MPIを使えば安価な４コアPCでも十分なリアルタイム符号化が可能であることも示した．

実装を行ったGPUはFermi以前のアーキテクチャであり，最新のKeplerアーキテクチャでは，メモリ管理の簡易化を含め，性能が大幅に向上しているため，最新のアーキテクチャに対応したアルゴリズムを検討することは必須である.また，マルチGPUでの実装とリアルタイムフラクタル動画符号化の実現は今後の課題である．

１０．　謝辞

本研究のプログラム実装に協力していただいた土屋俊貴,新井敬太の両氏に深く感謝いたします.

参考文献

１）Y. Fisher (ed.), Fractal Image Compression :Theory and Application, Springer-Verlag, 1994.

２）S.K.Alexander, E.R.Vrscay, S.Tsurumi, “An exami- nation of the statistical properties of domain-range block matching in fractal image coding”, Fractals in Engineering ’05 Proceedings(CD-ROM), 2005.

３）鶴見智, “反復関数系によるマルチメディアデータ符号化”, 群馬高専レビュー, No.19, pp.33-38, 2000.

４）高東建,徐粒,鶴見智, “SIMD 型超並列計算機によるフラクタル画像符号化,電子情報通信学会論文誌 D-II, Vol.J78-D-II,No.12,pp.1932-1934,1995.

５）U.Erra,” Toward Real Time Fractal Image Compression Using Graphics Hardware”, Advances in Visual Computing, Lecture Notes in Computer Science Volume 3804, pp 723-728, 2005.

６）Jacob Toft Pedersen, “”Paralell fractal compression for medical imaging”, PARALLEL COMPUTING FOR MEDICAL IMAGING AND SIMULATION, FALL, pp.1-6, 2010.

７）D.Chen and D.P.Singh. "Fractal video compression in OpenCL: An evaluation of CPUs, GPUs, and FPGAs as acceleration platforms." *ASP-DAC*. 2013.

８）伊藤智義編, GPU プログラミング入門-CUDA5 による実装-, 講談社, 2013.

９）NVIDIA 社ウェブサイト
http://www.nvidia.co.jp

Study on Parallel Computation in Fractal Coding -Implementation and Evaluation by GPU and MPI-

Satoshi TSURUMI

Fractal image coding is a block-based scheme that exploits the self-similarity hiding with an image. The main problem of fractal coding is the very high computing time needed to encode images. On the other hand, GPGPU (General Purpose computing on Graphic Processing Unit) attracts a great deal of attention, which is used for general-purpose computations like numerical calculations as well as graphic processing. In this study, we present an implementation of adaptive fractal coding on GPUs (Graphics Processing Units). We also evaluate the encoding performance comparing with MPI (Message Passing Interface). Finally, we discuss about a standard parallel computation algorithm of fractal coding.